A Versatile Intermediate Language for Answer Set Programming

Martin Gebser [1] and **Tomi Janhunen** [2] and **Max Ostrowski** [1] and **Torsten Schaub** [1] and **Sven Thiele** [1]
[1] Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam, Germany
[2] Helsinki University of Technology, Department of Information and Computer Science, P.O. Box 5400, FI-02015 TKK, Finland

Abstract

The attractiveness of Answer Set Programming (ASP) and related paradigms for declarative problem solving is considerably due to the availability of highly efficient yet easy-to-use implementations. A major driving force for the development and improvement of tools are standardized problem representations, for several reasons. First, they relieve developers from the burden of inventing their own input formats. Second, they establish interoperability between separate tools, allowing users to easily compare and exchange them without extensively converting their problem representations. Third, they facilitate the acquisition of problem descriptions from distinct sources, which is useful for benchmarking and assessment purposes. Historically, however, standards for representing logic programs, serving as inputs to ASP systems, were mainly dictated by the few available tools. In fact, there currently are two quasi standards, namely, the formats used by *lparse* and *dlv*, incompatible with each other. As a first step towards overcoming this deficiency, this work proposes an intermediate format for ground logic programs, intended for the representation of inputs to ASP solvers. The format is not designed to be a primary input language, given that ASP systems usually deploy a second component, called a grounder, to deal with the inputs provided by users. In view of this, our format is situated intermediate a grounder and a solver, guided by the example of grounder *lparse* and solver *smodels*, the latter marking the first among nowadays a variety of solvers processing the output of *lparse*. However, the output format of *lparse* has some decisive drawbacks, namely, its restrictive range and limited extensibility. We thus propose a new intermediate language, where our major design goals are flexibility in problem representation and easy extensibility to new language constructs.

Introduction

Answer Set Programming (ASP; (Baral 2003; Gelfond & Leone 2002; Marek & Truszczyński 1999; Niemelä 1999)) is a declarative approach to modeling and solving search problems, represented as logic programs. As illustrated in Figure 1, an ASP system usually deploys two components: a *grounder* and a *solver*. The input to an ASP system typically consists of a non-ground problem encoding and a ground problem instance. In such *uniform* encodings, the use of first-order variables reduces size and permits simpler, and therefore easier to write, logic programs. Furthermore, regarding the input language, several extensions have

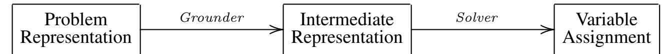

Figure 1: Basic Architecture of an ASP System

been proposed, like aggregates, cardinality and weight constraints, and optimize statements (Dell'Armi *et al.* 2003; Leone *et al.* 2006; Simons, Niemelä, & Soininen 2002). A grounder translates such a problem representation (typically a pair of an encoding and an instance) from the input language into a ground logic program, represented in a simplified, solver-readable form. Starting from a grounder's output, a solver then searches for answer sets, corresponding to solutions of the original problem. The most common solving approaches are based on the Davis-Putnam-Logemann-Loveland (DPLL; (Davis, Logemann, & Loveland 1962; Davis & Putnam 1960)) algorithm, like in *dlv* (Leone *et al.* 2006) and *smodels* (Simons, Niemelä, & Soininen 2002), or Conflict-Driven Clause Learning (CDCL; (Marques-Silva & Sakallah 1999; Mitchell 2005; Moskewicz *et al.* 2001)), e.g., used in *clasp* (Gebser *et al.* 2007a).

There currently is a single intermediate language accessible to ASP solvers, namely, the output format of grounder *lparse* (Syrjänen).[1] However, the format is not standardized and might thus change over different *lparse* versions, which is a delicate issue since no version information is included, e.g., for backward compatibility. The latter also makes ad hoc extensions of *lparse*'s output format intricate and error-prone.[2] Furthermore, the fact that *lparse*'s output format is designed to match *smodels*' internal data structures necessitates program transformations incurring a loss of structural information (Liu & Truszczyński 2005). We thus consider the restrictedness, on the one hand, and the limited extensibility, on the other hand, of *lparse*'s output format as serious drawbacks, making it unsuitable as a general standard.

This work proposes a new intermediate format, called *ASPils* ("ASP intermediate language standard"), for the use in-between grounders and solvers. Important design goals are:

[1] The *dlv* system uses an internal grounder that is directly coupled with the solver.

[2] For instance, the tool *dencode* (Janhunen) includes functionality for making conversions between the disjunctive rule output formats of old (up to version 1.0.14) and new versions of *lparse*.

- simplicity and efficiency in outputting and parsing;
- independence of grounder and solver implementations;
- support of the existing (input) language constructs (cf. (Dell'Armi *et al.* 2003; Leone *et al.* 2006) & (Syrjänen));
- support of version information, meta-information, and user comments; and
- flexibility and easy extensibility.

The development of *ASPils* is inspired by experiences made in related fields, such as Boolean Satisfiability (SAT), having standardized problem description languages, e.g., DIMACS format (DIMACS 1993).[3] For one, such standardized languages establish interoperability between solvers and further tools, for instance, tools generating solver inputs. As a concrete example in ASP, a standardized intermediate language might enable (arbitrary) solvers to process the output of *dlv*'s grounding component, and also *dlv*'s solving component to process the output of an external grounder. From a user's point of view, interoperability facilitates running different solvers, as a problem encoding written in the input language of a particular grounder could after grounding be processed by an arbitrary solver. Furthermore, a standardized intermediate language supported by all solvers would greatly foster ASP solver competitions, where in the past the different input formats of *dlv* and other solvers have been a major bottleneck (Gebser *et al.* 2007b). In fact, as a secondary benefit, a common language eases collecting challenging benchmarks from distinct sources and might thus push the further development of ASP solvers, like it has been experienced in SAT. To this end, this paper introduces *ASPils* and illustrates its potential usage on examples; full details are provided in (Gebser *et al.* 2008a).

In order to put this document in perspective, let us stress that *ASPils* is proposed as a standard for the transmission of ground logic programs from grounders to solvers. At this stage, our proposal aims at recording the language constructs currently supported by *lparse*-based solvers as well as *dlv*'s solving component and at integrating them into a common framework, also anticipating future extensions to a certain extent. Of course, an input format for ASP solvers can only turn into a standard if it is widely supported and used in practice, which requires a community effort, especially, from ASP system developers. Our proposal of *ASPils* thus aims at providing a starting point for a community-wide discussion of a standardized input format for ASP solvers. Even if such a standard is successfully established, it will not instantly abolish all differences and peculiarities of ASP systems. For instance, grounders and integrated ASP systems may further (have to) use proprietary input languages, and any modular (Oikarinen & Janhunen 2006), incremental (Gebser *et al.* 2008b), or even a system not following the computational pattern shown in Figure 1 might not be readily supplied with an appropriate input format. However, before succeeding to standardize the simplest element in the workflow of ASP systems, namely, the intermediate language used in-between a grounder and a solver, any attempts to standardize more diversified matters would most likely be prone to fail. Hence, even though the scope of *ASPils* is limited to an intermediate representation according to Figure 1, we think that it may initiate a worthwhile discussion on language standards in ASP.

[3] See (Janhunen 2007; Gebser *et al.* 2008a) for detailed discussions of intermediate formats.

General Design of *ASPils*

We now briefly describe the design decisions underlying *ASPils*, the new intermediate language for ASP we propose here. Our global goal is to specify a language that has the potential to become a standard format for inputs to ASP solvers. Thus, we have to respect that different ASP solvers support different language constructs, e.g., *dlv* deals with aggregates (Dell'Armi *et al.* 2003) and *smodels* with extended rules (Simons, Niemelä, & Soininen 2002). In order to reflect this diversity, our language must be general and solver-independent. Furthermore, it is impossible to foresee language constructs that might evolve in the future. Hence, extensibility of the language is an important issue. We thus include a *version number* in problem descriptions of *ASPils*. In addition, *normal forms* are used to specify language fragments. Their main purpose is to reflect different capabilities of solvers, which are thus enabled to check whether a problem description is appropriate before processing it further.

The body of *ASPils* consists of *entries*, mainly defining *objects* of particular *types*. The idea is similar to the output format of *lparse* (Syrjänen), using several rule types. However, *ASPils* goes further than *lparse* by not restricting types to rules, rather, all entities in a ground logic program, e.g., atoms, conjunctions, disjunctions, etc., are objects having a type. An advantage of this is that complex structures occurring in a program, e.g., conjunctions of cardinality and weight constraints, can be represented in a modular and structure-preserving way. In contrast, *lparse* would have to introduce new atoms and rules to represent complex structures in its restrictive output format. Another advantage of types is the easy embedding of new language constructs, as it only requires the definition of a type identifier and a syntax for objects of the type. To avoid clashes of custom type identifiers, we use numbers consisting of a "major" and a "minor" slot (similar to IP addresses), where the major slot ought to be related to a research group defining the type. If newly introduced types turn into a standard, they can be integrated into *ASPils* via an additional normal form or even a new language version.

As mentioned above, every *object* occurring in a problem description has an associated type. In addition, each object has a unique ID, that is, a positive integer, to refer to the object. This makes the language modular because, on the syntactic level, other objects make use only of the ID of an referenced object, but not of its internal structure. Hence, if new language constructs are introduced, their objects can immediately be used within available structures, without needing to exchange them. A second potential benefit of object IDs is the possibility to re-use them if the same object has multiple occurrences in a ground logic program, thus compacting the representation. Note that this accounts

for ordinary propositional atoms as well as for non-atomic structures, such as conjunctions and disjunctions.

On the technical level, our language shall be easy to parse, independent of system environments, and resistant against potential parsing errors. To this end, we use a numerical text format and number `0` as an explicit delimiter for entries. As `0` can also occur within an entry (not as delimiter), each entry must specify its number of consecutive (numeric and/or symbolic) parameters directly after its type. This shall enable the correct syntactic decomposition of *ASPils* sentences, even without recognizing the contents, and it deliberately introduces a layer of redundancy in order to avoid parsing errors. Also note that most parameters occurring in entries are numeric and thus represented by integers, which should facilitate their recognition by solvers. In particular, negative integers are used to denote the default negations of objects having the absolute values as their IDs. Of course, the use of numbers makes *ASPils* less human-readable, but human-readability is not one of our design goals anyway. Symbolic information can still be included, most likely, for defining atom names, but usually such information needs not be interpreted by solvers. Furthermore, arbitrary meta-information as well as user comments can be provided using dedicated types. Note that comments are the only kind of objects not having an ID, as they ought to be ignored by solvers. In contrast, meta-information may be exploited by solvers, but it should not be mandatory for solvers to recognize it. We do not suggest any kind of meta-information, but information like whether a logic program is tight (Fages 1994; Erdem & Lifschitz 2003) or whether it has an answer set might be useful for particular purposes.

Language Description

This section describes the elementary constituents of our proposed intermediate language *ASPils*, where we focus on intuitions and examples. The formal specification of *ASPils* can be found in (Gebser *et al.* 2008a). Below, *italic* and `typewriter` fonts indicate non-terminals and terminals, respectively, in the grammar of *ASPils*.

Header

Every sentence of *ASPils* starts with a *header*.[4] E.g., header

```
1 3 1 3 0 0
```

consists of a `1` indicating the *type header*, a `3` providing the number of parameters before the delimiter, the second `1` stating that this is the first *version* of *ASPils*, the second `3` indicating conformance to *normal form* "SModels" (introduced below), a `0` stating that there are no *additional headers*, and the second `0` delimiting the header. Note that language version `1` of *ASPils*, defined in (Gebser *et al.* 2008a), does not specify any additional headers, hence, their number will always be `0`. The pattern that a type number is followed by the number of parameters before delimiter `0` recurs in each of the types described below, while the parameters themselves are specific.

[4]Only *comments* are allowed before *header* and after *object eof*.

End Of File

Every sentence of *ASPils* is terminated with an occurrence of *object eof*,[4] looking as follows:

```
0 0 0 .
```

The first `0` indicates the object's type, the second its number of parameters, and the third one delimits it. (The full stop sign "." is part of the surrounding text, but not of *ASPils*.)

Entries

In-between the header and end of file, a ground logic program is specified by entries, providing meta-information, comments, or defining elements of the program at hand. Each *entry* starts with its type number, an up to eight digits long hexadecimal cipher. The first four digits denote the research group that has developed the type; for the core types described below, the four leading digits are zeros and can thus be omitted. The last four digits of a type number must not evaluate to zero (reserved for *object eof*). Each type number is followed by the number of consecutive parameters before the *entry* is delimited by an occurrence of `0`. Each entry type imposes particular parameters, that is, the slots specified in the grammar (Gebser *et al.* 2008a) must be filled with terminals.

Meta-Information. Objects of type `2` can be used to provide meta-information. Although we do not suggest any particular meta-object, the following one is syntactically valid:

```
2 3 42 "tight program" 23 0 .
```

Among the `3` parameters of the entry, `42` is the *object ID*, `"tight program"` is a *safe verbal*, that is, a list of strings and whitespaces enclosed in double quotes, and `23` is the single element of a list of *meta-option*s (that is, *integer*s). Meta-information may be exploited by solvers, but any such information should not affect the semantics of the specified ground logic program, so that it is admissible for solvers to simply ignore unrecognized meta-objects.

Comments. Comments of type `3` do not define any object (i.e., they do not have an *object ID* as parameter). An example comment looks as follows:

```
3 1 "grounded by GrinGo version 2" 0 .
```

As *comment*s are not associated with an ID, they cannot be referenced by objects defined in an *ASPils* problem description. In fact, comments are understood to be completely up to user information, such as the author of a problem description or the grounder that generated it. Unlike meta-information, comments must always be ignored by solvers.

Atoms. Objects of type `4` define atoms. E.g., consider:

```
4 2  8   p(a,1)        0
4 5 15 "-p(a,1)" 1 2 8 0 .
```

The first entry specifies that *object ID* `8` stands for an *atom* whose name is `p(a,1)`, and the second entry defines *ob-*

ject ID `15` to represent another atom with name `-p(a,1)`.[5] Furthermore, *atom option* `1`, provided for `-p(a,1)`, declares the atom as hidden, that is, the atom name shall be suppressed in the output of an ASP solver. We include this option in order to reflect the effect of hide declarations in the input language of *lparse* (Syrjänen). However, while *lparse* suppresses atom names by not including them in the symbol table, we choose to keep the symbolic names of atoms and to signal their hidden nature via an option. In this way, it stays possible to recover a symbolic representation from the intermediate format, as it is done by tool *lplist* (Janhunen) for *lparse*'s output format using the symbolic information still available there. The second *atom option* `2` for `-p(a,1)` declares the atom to be the classical negation of the object with ID `8`, viz., of atom `p(a,1)`. Classical negation is understood in the sense of (Gelfond & Lifschitz 1991). In our example, it means that atoms `p(a,1)` and `-p(a,1)` cannot jointly belong to a stable model of the program at hand.

Rules, Facts, and Integrity Constraints. In ASP, a logic program is a set of rules, each rule consisting of a head and a body. Either the head or the body may be constant, in which case the rule is called a fact or an integrity constraint, respectively. Let us consider the following example logic program containing a rule, a fact, and an integrity constraint:

```
-p(a,1) :- not p(a,1).
 p(a,1).
        :- -p(a,1).
```

Reusing *object ID*s `8` and `15` for `p(a,1)` and `-p(a,1)`, respectively, corresponding entries in *ASPils* are as follows:

```
5 3 55 15 -8 0
6 2 66  8    0
7 2 77    15 0 .
```

Here, type numbers `5`, `6`, and `7` indicate that the objects with IDs `55`, `66`, and `77` are a *rule*, a *fact*, and an *integrity constraint*, respectively. Note that `-8` in the first entry refers to the default negation of the atom with ID `8`, viz., of atom `p(a,1)`. Furthermore, observe that the second entry specifies only a head *literal* and the third one only a body *literal*, while the rule defined by the first entry contains both a head and a body *literal*. The choice of introducing three different types is motivated by the goal of not imposing any hard-wired assumptions on the structure of heads and bodies, while still being able to identify the role of particular *literal*s. Importantly, workarounds such as the `_false` atom, introduced by *lparse* (Syrjänen) as the head of integrity constraints, ought to be avoided. Due to the generic design of entries for rules, facts, and integrity constraints, allowing to refer to arbitrary and possibly default negated objects, there are no restrictions on the structure of heads and bodies (rules might even reference each other) a priori. Below, the issue of ensuring certain formats is dealt with via normal forms.

[5]The name of the second atom must be provided as a *safe verbal*, enclosed in double quotes. Double quotes may only be omitted for *atom name*s starting with a *letter*. Due to this requirement, the first symbols of *atom name*s and *integer*s become unambiguous.

Conjunctions and Disjunctions. In order to express more complex rules than the ones given above, entries may specify conjunctions and disjunctions of *literal*s. For instance,

```
8 3 89 -8 -15 0
```

defines an object with ID `89` as the *conjunction* of the default negations of the objects with IDs `8` and `15`. Similarly,

```
9 3 98 8 15 0
```

defines a *disjunction* of the objects with IDs `8` and `15`. Assuming that ID `8` stands for atom `p(a,1)` and `15` for `-p(a,1)`, the entries

```
7 2 78 89 0
6 2 67 98 0
```

describe the following integrity constraint and disjunctive fact:

```
:- not p(a,1), not -p(a,1).
p(a,1) | -p(a,1).
```

Finally, note that the normal forms described below restrict conjunctions to occur in bodies of rules and disjunctions to being used in rule heads.

Default Negation. Programs in "canonical form" (Lee 2005; Lifschitz, Tang, & Turner 1999) permit double (default) negation of atoms. Rather than permitting multiple occurrences of "−" at the beginning of a *literal* (which would make *literal*s and *integer*s syntactically different), we introduce entries defining default negation objects for representing nested negation. This allows us to express a "choice"

```
p(a,1) :- not not p(a,1).
```

in terms of the following entries:

```
a 2 44   -8 0
5 3 45 8 44 0 .
```

Observe that the object defined by the first entry stands for the *default negation* of *literal* `-8` and, thus, for the double negation of the object with ID `8`, viz., of atom `p(a,1)`. Finally, note that, under answer set semantics, rules using an atom and those using its double negation are not necessarily equivalent (Lifschitz, Tang, & Turner 1999), so that double negation constitutes a proper syntactic feature.

Cardinality and Weight Constraints. Cardinality and weight constraints (Simons, Niemelä, & Soininen 2002; Syrjänen) permit expressing conditions on sets of literals, that is, true literals can be counted or their weights can be summed up in order to compare the result against a lower and an upper bound. Letting *object ID*s `1`, `2`, `3`, and `4` stand for atoms `a`, `b`, `c`, and `d`, we can represent the expressions

```
2{a, b, not c, not d}3
-1[a=-2, b=1, not c=3, not d=-4]2
```

in *ASPils* as follows:

```
b  7 5  2 3 1 2 -3 -4           0
c 11 6 -1 2 1 2 -3 -4 -2 1 3 -4 0 .
```

Here, the *object ID*s `5` and `6` of the *cardinality* and *weight constraint*, respectively, are immediately followed by their lower and upper bounds. Afterwards, the *literal*s are provided, and for weight constraint `6`, also a list of *weight*s (exactly one weight per literal). The primary constituents of a

logic program, viz., *rules*, *facts*, and *integrity constraints*, can incorporate *cardinality* and *weight constraints* just like *atoms*, simply by referencing their IDs directly or indirectly through literals. Observe that, in general, we allow for upper bounds as well as for negative weights, both of which can occur in the input language, but not in the output language, of *lparse*. In fact, *lparse* performs a number of transformations and introduces new atoms to remove them. Some of the normal forms below impose similar restrictions, thus, specifying *lparse*-like fragments of *ASPils*. In such fragments, only *trivial upper bounds* are permitted, obtained by summing up all (positive) weights, where weight `1` is used for the literals of cardinality constraints.

Weighted Literals. *Weighted literals* are auxiliary concepts, devised for the use with aggregates and optimization statements (see below), thus, establishing a uniform way of referencing objects (simple or complex ones) evaluating to numbers. For instance, we may associate *weights* to *literals*, as in the above weight constraint, via the following entries:

```
d 3 11  1 -2 0
d 3 12  2  1 0
d 3 13 -3  3 0
d 3 14 -4 -4 0 .
```

Aggregates. We adopt the aggregates supported by *dlv* (Dell'Armi *et al.* 2003), allowing for five operations, viz., *count*, *sum*, *max*, *min*, and *times*. While *count* applies to Boolean operands, that is, to *literals*, the other four aggregates operate on numerical values, thus, they require *object IDs* rather than *literals* as parameters. Reusing atoms `a`, `b`, `c`, and `d` as well as weighted literals as specified above, we may define a *count* and a *sum* aggregate as follows:

```
e 5 21  1  2 -3 -4 0
f 5 22 11 12 13 14 0 .
```

Observe that *count* applies to (possibly negative) *literals*, while *dlv*'s aggregates are restricted to atoms. As such a restriction does not significantly simplify dealing with aggregates, we do not adopt it here, and the *weighted literals* used by *sum* may also apply to negative *literals* (as it is the case for the objects with IDs `13` and `14`). However, in order to reasonably apply an aggregate, operands must have appropriate types, being an issue to the normal forms below.

Operators. Arithmetic comparison operators can be applied to *weighted literals* and *aggregates* in order to retrieve Boolean values from them. Thus, *operators* can be referenced by *rules*, *facts*, and *integrity constraints* in the same way as *atoms*. We provide two kinds of operators: (binary) *operators* of type in-between `13` and `17` can be used to compare the numerical values of two *objects* with one another, while *unary operators* of type in-between `18` and `1c` allow for comparing an *object*'s numerical value to an *integer*. Both kinds of *operators* support the following comparison operations: *eq* ("equal"), *leq* ("less or equal"), *lt* ("less than"), *geq* ("greater or equal"), and *gt* ("greater than"). For instance, the following entry describes the application of the (binary) *operator eq* to the aggregates with IDs `21` and `22` as defined above:

```
13 3 31 21 22 0 .
```

An application of *unary operator leq* to the aggregate with ID `22` and integer `0` can be specified as follows:

```
19 3 91 22 0 0 .
```

Finally, note that current ASP solvers do not support (binary) *operators* of type in-between `13` and `17`, hence, they will not be permitted by the normal forms below.

Optimization. Amongst the most common optimization techniques in ASP are "minimize statements" (Simons, Niemelä, & Soininen 2002) supported by *smodels* and "weak constraints" (Leone *et al.* 2006) supported by *dlv*. In order to reflect them, we introduce an *optimize* object whose underlying *strategy* is minimization of (arbitrarily many) objective functions of distinct priorities, the priorities forming a strict total order. Other strategies than lexicographic ordering of objective functions, e.g., Pareto optimality, would also be possible but are currently not in use, so we do not (yet) consider them. In what follows, we detail how "minimize statements" and "weak constraints" can be represented in *ASPils*.

Minimize Statements of *smodels*: Assume that an input program provided to *lparse* contains two minimize statements (in order):

```
minimize[not a, b, c].
minimize[a=4, not b=3, c=2].
```

The first statement expresses that a minimum number of its literals should be true, while the second one is about minimizing the sum of weights of true literals. The corresponding objective functions are expressed by a *count* and a *sum* aggregate (over *weighted literals*), respectively, where we assume IDs `1`, `2`, and `3` for atoms `a`, `b`, and `c`:

```
d 3 11  1  4    0
d 3 12 -2  3    0
d 3 13  3  2    0
e 4 21 -1  2  3 0
f 4 22 11 12 13 0 .
```

Note that the *count* aggregate with ID `21` gives the objective function minimized by the first statement over literals, and the *sum* aggregate with ID `22` takes the weights provided in the second minimize statement into consideration. We are now ready to define a single *optimize* object that incorporates both aggregates:

```
1d 4 43 1e 22 21 0 .
```

The type of this object is `1d`, `4` the number of parameters, and `43` the ID. Furthermore, `1e` specifies the lexicographic optimization strategy, which is the only *strategy* included in the first version of *ASPils*. Finally, minimizing the numerical value of the *sum* aggregate with ID `22` takes higher priority than minimizing the value of the *count* aggregate with ID `21`. This priority, reverse to the order of minimize statements in the input, is indeed applied by *smodels*. While *smodels* derives the (reverse) priorities of minimize statements implicitly from the order in the input, in the *ASPils* representation, priorities are explicit because the objective functions to be minimized are combined within an *optimize* object.

Weak Constraints of *dlv*: We start with an example. Consider the following weak constraints:

```
:~ a, not b. [1:2]
:~ not c. [1:1]
:~ b, c. [2:1]
```

Note that the numbers in brackets describe weights and levels. As level 2 of the first weak constraint is greater than 1, the first weak constraint is of higher priority than the second and the third one. Among the last two weak constraints, the priority of the third one is greater simply because its weight 2 is greater than 1. In order to express the non-singleton bodies of weak constraints, we define *conjunction*s as follows:

```
8 3 81 1 -2 0
8 3 83 2  3 0 .
```

The *conjunction* with ID 81 stands for the body of the first weak constraint, and the one with ID 83 for the body of the third weak constraint. We can now proceed by defining *weighted literal*s, one per weak constraint:

```
d 3 11 81 1 0
d 3 21 -3 1 0
d 3 22 83 2 0 .
```

Observe that the *weight* of each weak constraint is the weight given in the input. Finally, we use a *sum* aggregate for multiple weak constraints at the same level and define an *optimize* object as follows:

```
f  3 24    21 22 0
1d 4 35 1e 11 24 0 .
```

The last entry expresses that we minimize weights starting with the weak constraint at level 2 and, secondarily, for the weak constraints at level 1.

We now provide a general scheme of how to represent multiple weak constraints of distinct levels in *ASPils*. Consider the following weak constraints ordered by their levels l_j, where we assume $l_i > l_j$ if $i < j$ in order to respect level priorities:

$$
\begin{array}{ccc}
:\tilde{}\ body_{1_1}.\ [w_{1_1} : l_1] & \ldots & :\tilde{}\ body_{n_1}.\ [w_{n_1} : l_1] \\
& \vdots & \\
:\tilde{}\ body_{1_m}.\ [w_{1_m} : l_m] & \ldots & :\tilde{}\ body_{n_m}.\ [w_{n_m} : l_m]\ .
\end{array}
$$

In *ASPils*, the bodies $body_{1_1}, \ldots, body_{n_m}$ of weak constraints can be defined by *literal*s either over *atom*s (for singleton bodies) or over *conjunction*s, which can be defined as usual. Thus, we keep notation $body_{i_j}$ in the following *weighted literal*s:

$$
\begin{array}{lllll}
\texttt{d 3} & x_{1_1} & body_{1_1} & w_{1_1} & \texttt{0} \\
\vdots & & & & \\
\texttt{d 3} & x_{n_1} & body_{n_1} & w_{n_1} & \texttt{0} \\
\vdots & & & & \\
\texttt{d 3} & x_{1_m} & body_{1_m} & w_{1_m} & \texttt{0} \\
\vdots & & & & \\
\texttt{d 3} & x_{n_m} & body_{n_m} & w_{n_m} & \texttt{0}\ .
\end{array}
$$

The *weighted literal*s defined above are similar to those used in the representation of minimize statements. In fact, it makes no difference in *ASPils* whether they refer to *atom*s or to *conjunction*s. The next step of adding level-wise *sum* aggregates and a single *optimize* object is similar to minimize statements:

$$
\begin{array}{llllll}
\texttt{f} & n_1{+}1 & s_1 & x_{1_1}\ \ldots\ x_{n_1} & \texttt{0} & \\
\vdots & & & & & \\
\texttt{f} & n_m{+}1 & s_m & x_{1_m}\ \ldots\ x_{n_m} & \texttt{0} & \\
\texttt{1d} & m{+}2 & o & \texttt{1e}\ \ s_1\ \ldots\ s_m & \texttt{0}\ . &
\end{array}
$$

The given embeddings in *ASPils* indicate that minimize statements and weak constraints are handled likewise, using *weighted literal*s, *sum* aggregates (sometimes, simpler constructs are sufficient), and an *optimize* object.

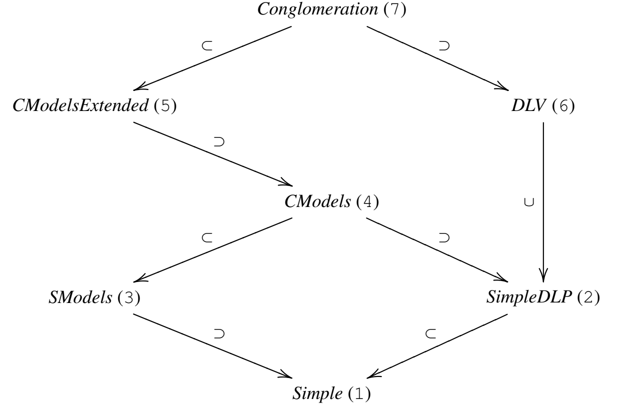

Figure 2: Normal Form Hierarchy

Normal Forms

This section describes seven normal forms corresponding to different language fragments handled by existing ASP solvers. The normal forms stand in a hierarchy, as shown in Figure 2. Each of the normal forms is identified via a corresponding number, given in parentheses in Figure 2, to be provided within the *header* of a problem description in *ASPils*. The full specification of the normal forms presented below can be found in (Gebser *et al.* 2008a). In particular, admissible object types and reference relationships are defined for each normal form in turn. In what follows, we focus on their main features and provide examples.

Normal Form *Simple*

This *normal form* corresponds to the input language used in the *SCore* category of the ASP system competition (Gebser *et al.* 2007b). It allows for representing ground normal logic programs without any extended constructs (like aggregates, etc.). For example, consider the following input program:

```
a :- not b.
b :- not a.
:- b.
c :- d, not b.
d.
#hide a.
```

This program can be represented in *ASPils* as follows:[6]

[6]We indicate the meanings of *object*s in comments preceding their definitions.

```
3 1 "header, language version =: 1,
            normal form =: 1" 0
1 3 1 1 0 0
3 1 "a =: 1 and hidden,
    b =: 2, c =: 3, d =: 4" 0
4 3 1 a 1 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "(a :- not b.) =: 5" 0
5 3 5 1 -2 0
3 1 "(b :- not a.) =: 6" 0
5 3 6 2 -1 0
3 1 "(:- b.) =: 7" 0
7 2 7 2 0
3 1 "(d, not b) =: 8" 0
8 3 8 4 -2 0
3 1 "(c :- d, not b.) =: 9" 0
5 3 9 3 8 0
3 1 "(d.) =: 10" 0
6 2 10 4 0
3 1 "end of file" 0
0 0 0 .
```

Note that the above representation is not unique, for instance, we could have assigned different *object IDs* or changed the order of entries.

Normal Form *SimpleDLP*

This *normal form*, corresponding to the input language used in the $SCore^{\vee}$ category of the ASP system competition (Gebser *et al.* 2007b), extends normal form "Simple" by allowing *disjunctions* over *atoms* to occur in heads of *rules* and *facts*. E.g., consider the following disjunctive program:

```
a | b.
b | c | d :- a, not d.
```

This program can be represented in *ASPils* as follows:

```
3 1 "header, language version =: 1,
            normal form =: 2" 0
1 3 1 2 0 0
3 1 "a =: 1, b =: 2, c =: 3, d =: 4" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "(a | b) =: 5" 0
9 3 5 1 2 0
3 1 "(a | b.) =: 6" 0
6 2 6 5 0
3 1 "(b | c | d) =: 7" 0
9 4 7 2 3 4 0
3 1 "(a, not d) =: 8" 0
8 3 8 1 -4 0
3 1 "(b | c | d :- a, not d.) =: 9" 0
5 3 9 7 8 0
3 1 "end of file" 0
0 0 0 .
```

As in the previous subsection, this representation in *ASPils* is not unique.

Normal Form *SModels*

This *normal form* is inspired by the input language of solver *smodels* (Simons, Niemelä, & Soininen 2002), that is, it extends "Simple" by *cardinality* and *weight constraints* as well as *optimize* objects. Note that the *weights* used in *weight constraints* and *weighted literals* have to be non-negative. Furthermore, the upper bounds of *cardinality* and *weight constraints* must be trivial, that is, they cannot be smaller than the number of literals or the sum of weights, respectively, in a constraint. If a *cardinality constraint* occurs as the head of a *rule* or *fact*, its lower bound must also be trivial, viz., it must be 0, while *weight constraints* are not permitted as heads. Finally, note that *weighted literals* as well as *count* and *sum* aggregates may only be used in combination with an *optimize* object, but not as a part of a *rule*, a *fact*, or an *integrity constraint*. Let us consider the following program:

```
{a, b}.
c :- a, not b.
:- 3[a=2, b=1, not c=2].
minimize[not a=1, not b=2, c=2].
```

The following is a possible representation in *ASPils*:

```
3 1 "header, language version =: 1,
            normal form =: 3" 0
1 3 1 3 0 0
3 1 "a =: 1, b =: 2, c=: 3" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
3 1 "{a, b} =: 4" 0
b 5 4 0 2 1 2 0
3 1 "({a, b}.) =: 5" 0
6 2 5 4 0
3 1 "(a, not b) =: 6" 0
8 3 6 1 -2 0
3 1 "(c :- a, not b.) =: 7" 0
5 3 7 3 6 0
3 1 "3[a=2, b=1, not c=2] =: 8" 0
c 9 8 3 5 1 2 -3 2 1 2 0
3 1 "(:- 3[a=2, b=1, not c=2].) =: 9" 0
7 2 9 8 0
3 1 "(not a=1) =: 10, (not b=2) =: 11,
        (c=2) =: 12" 0
d 3 10 -1 1 0
d 3 11 -2 2 0
d 3 12  3 2 0
3 1 "sum[not a=1, not b=2, c=2] =: 13" 0
f 4 13 10 11 12 0
3 1 "(minimize[not a=1, not b=2, c=2].)
     =: 14" 0
1d 3 14 1e 13 0
3 1 "end of file" 0
0 0 0 .
```

Normal Form *CModels*

This *normal form* is closely related to the input language of solver *cmodels* (Giunchiglia, Lierler, & Maratea 2006; Lierler 2005), basically, augmenting "SModels" normal form with *disjunctions* in heads of *rules* and *facts*.[7]

Normal Form *CModelsExtended*

This *normal form* is derived from "CModels" by dropping some restrictions. Non-trivial upper bounds are permitted for *cardinality* and *weight constraints*. Furthermore, both of them can occur with non-trivial bounds as heads of *rules* and

[7] While *cmodels* does not process minimize statements, they can be expressed in "CModels" via *optimize* objects.

*fact*s. Finally, negative *weight*s can be used within *weight constraint*s and *weighted literal*s being subject to *optimize* objects. The following program uses these extra features:

```
0[a=1, b=-1]0 :- 0[c=-1, d=1]0.
0[c=1, d=-1]0 :- 0[a=-1, b=1]0.
minimize[not a=-1, not b=2, c=-2, d=1].
```

This program can be represented in *ASPils* as follows:

```
3 1 "header, language version =: 1,
                normal form =: 5" 0
1 3 1 5 0 0
3 1 "a =: 1, b =: 2, c =: 3, d =: 4" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "0[a=1, b=-1]0 =: 5" 0
c 7 5 0 0 1 2 1 -1 0
3 1 "0[c=-1, d=1]0 =: 6" 0
c 7 6 0 0 3 4 -1 1 0
3 1 "(0[a=1, b=-1]0 :- 0[c=-1, d=1]0.)
      =: 7" 0
5 3 7 5 6 0
3 1 "0[c=1, d=-1]0 =: 8" 0
c 7 8 0 0 3 4 1 -1 0
3 1 "0[a=-1, b=1]0 =: 9" 0
c 7 9 0 0 1 2 -1 1 0
3 1 "(0[c=1, d=-1]0 :- 0[a=-1, b=1]0.)
      =: 10" 0
5 3 10 8 9 0
3 1 "(not a=-1) =: 11, (not b=2) =: 12,
          (c=-2) =: 13,     (d=1) =: 14" 0
d 3 11 -1 -1 0
d 3 12 -2  2 0
d 3 13  3 -2 0
d 3 14  4  1 0
3 1 "sum[not a=-1, not b=2, c=-2, d=1]
      =: 15" 0
f 5 15 11 12 13 14 0
3 1 "(minimize[not a=-1, not b=2, c=-2,
                        d=1].) =: 16" 0
1d 3 16 1e 15 0
3 1 "end of file" 0
0 0 0 .
```

Normal Form *DLV*

This *normal form* is inspired by the input language of solver *dlv* (Dell'Armi *et al.* 2003; Leone *et al.* 2006). Hence, it allows for *disjunction*s over *atom*s in heads of *rule*s and *fact*s. Furthermore, aggregates may be used in bodies, under the proviso that all referenced *weighted literal*s have non-negative *weight*s.[8] As *dlv* does not deal with cardinality and weight constraints, we exclude *cardinality* and *weight constraint*s from "DLV" normal form. (However, *cardinality* and *weight constraint*s can equivalently be expressed in terms of *count* and *sum* aggregates, respectively.) Finally, weak constraints (Leone *et al.* 2006) can be represented using *optimize* objects. For illustration, consider program:

```
a | b | c.
d :- sum[a=1, b=1, c=2] >= 2.
:~ d, not b. [1:2]
:~ a. [2:1]
:~ not c. [1:1]
```

The following is a representation of this program in *ASPils*:

```
3 1 "header, language version =: 1,
                normal form =: 6" 0
1 3 1 6 0 0
3 1 "a =: 1, b =: 2, c =: 3, d =: 4" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "(a | b | c) =: 5" 0
9 4 5 1 2 3 0
3 1 "(a | b | c.) =: 6" 0
6 2 6 5 0
3 1 "(a=1) =: 7, (b=1) =: 8, (c=2) =: 9" 0
d 3 7 1 1 0
d 3 8 2 1 0
d 3 9 3 2 0
3 1 "sum[a=1, b=1, c=2] =: 10" 0
f 4 10 7 8 9 0
3 1 "(sum[a=1, b=1, c=2] >= 2) =: 11" 0
1b 3 11 10 2 0
3 1 "(d :- sum[a=1, b=1, c=2] >= 2.)
      =: 12" 0
5 3 12 4 11 0
3 1 "(d, not b) =: 13" 0
8 3 13 4 -2 0
3 1 "((d, not b)=1) =: 14, (a=2) =: 15,
          (not c=1) =: 16" 0
d 3 14 13 1 0
d 3 15  1 2 0
d 3 16 -3 1 0
3 1 "sum[a=2, not c=1] =: 17" 0
f 3 17 15 16 0
3 1 "(minimize[a=2, not c=1].
      minimize[(d, not b)=1].) =: 18" 0
1d 4 18 1e 14 17 0
3 1 "end of file" 0
0 0 0 .
```

Normal Form *Conglomeration*

This *normal form* is the most general one provided here. It results from "CModelsExtended" and "DLV" by dropping some restrictions of the latter, that is, *unary operator*s and *aggregate*s may occur in the heads of *rule*s and *fact*s, and negative *weight*s are allowed within *weighted literal*s. Furthermore, we include *default negation* objects on negative *literal*s over *atom*s in order to account for double negation. Though double negation is a syntactical feature that increases neither computational complexity nor technical difficulties of ASP solving, somewhat astonishingly, it is currently not supported by any ASP solver nor by accompanying grounders. This is why *default negation* objects were not permitted in previous normal forms. The following program, comprising a single rule, uses the additional features:

```
sum[a=1, b=1, c=1, d=-2] == 0 :-
  2[a=-1, not b=2, not c=3]3, not not d.
```

This program can be represented in *ASPils* as follows:

```
3 1 "header, language version =: 1,
                normal form =: 7" 0
1 3 1 7 0 0
3 1 "a =: 1, b =: 2, c =: 3, d =: 4" 0
```

[8] The *dlv* solver requires logic programs to be "aggregate-stratified" (Dell'Armi *et al.* 2003), which is not reflected herein.

```
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "(a=1) =: 5, (b=1) =: 6, (c=1) =: 7,
     (d=-2) =: 8" 0
d 3 5 1  1 0
d 3 6 2  1 0
d 3 7 3  1 0
d 3 8 4 -2 0
3 1 "sum[a=1, b=1, c=1, d=-2] =: 9" 0
f 5 9 5 6 7 8 0
3 1 "(sum[a=1, b=1, c=1, d=-2] == 0)
      =: 10" 0
18 3 10 9 0 0
3 1 "2[a=-1, not b=2, not c=3]3 =: 11" 0
c 9 11 2 3 1 -2 -3 -1 2 3 0
3 1 "(not not d) =: 12" 0
a 2 12 -4 0
3 1 "(2[a=-1, not b=2, not c=3]3,
       not not d) =: 13" 0
8 3 13 11 12 0
3 1 "(sum[a=1, b=1, c=1, d=-2] == 0 :-
         2[a=-1, not b=2, not c=3]3,
         not not d.) =: 14" 0
5 3 14 10 13 0
3 1 "end of file" 0
0 0 0 .
```

Discussion and Outlook

We have described language version `1` of *ASPils* ("ASP intermediate language standard"); full details can be found in (Gebser *et al.* 2008a). The primary motivation for this work is making a step towards a standard input language for ASP solvers to be generated by grounders, for which we propose *ASPils*. The major design goals of *ASPils* are generality, by supporting language constructs processed by existing ASP grounders and solvers, and extensibility, by using an object-based approach and including version information. For solvers, parsing a problem description in *ASPils* should still be reasonably simple, thus, *ASPils* defines a numerical format not intended to be manually written by users. However, *ASPils* also provides means to specify symbolic information, enabling the reconstruction of a human-readable format. Beyond that, via comments and meta-information, arbitrary contents can be included in a problem description without disturbing solvers. The current proposal of *ASPils* is a good response to the recommendations presented in (Janhunen 2007) as regards extensibility and support for comments as well as symbolic information.

As a proof of concept, we are currently working on a new version of grounder *gringo* (Gebser, Schaub, & Thiele 2007) able to output *ASPils* format and also on an *ASPils* front-end for solver *clasp* (Gebser *et al.* 2007a). In the course of this, we take advantage of the generic design of *ASPils* allowing us to preserve the structure of ground logic programs. For instance, *gringo* can output cardinality and weight constraints specifying both a lower and a non-trivial upper bound, and such constraints can occur both in the bodies and in the heads of rules. In contrast, in *lparse*'s output format (Syrjänen), upper bounds (and in rule heads, also lower bounds) have to be compiled away, introducing additional atoms and rules. Such structure-degrading transformations are performed by *lparse* in order to match the internal data structures of *smodels* (Simons, Niemelä, & Soininen 2002), and in the past, tools (Liu & Truszczyński 2005) were particularly developed to undo such transformations. As regards grounding, we think that the two tasks of a grounder are, first, substituting constants for variables in an input program and, second, presenting the grounding result to a solver in some basic format that is easy to parse. Beyond these two tasks, a grounder should keep the input program intact in order to be solver-independent and to abolish the need of applying structure-restoring tools. In particular, introducing additional atoms during the grounding phase ought to be avoided, as it is very likely to spoil the desired equivalence between the input program and the grounding result.

In long-term, we hope that our proposal of an intermediate language standard leads to the establishment of a common input format for ASP solvers, comparable to the role of DIMACS format (DIMACS 1993) in SAT. On the one hand, it would make ASP more user-friendly if solvers could be interchanged without redoing problem encodings, given that the two main input languages, the one of *dlv* (Leone *et al.* 2006) and the one of *lparse* (Syrjänen), are incompatible with each other. A common intermediate language would enhance the interoperability of other auxiliary tools as well. Similarly, the assessment of ASP solvers would be greatly facilitated, for instance, in future ASP solver competitions. On the other hand, the non-availability of a standardized intermediate language (as *dlv* does not supply an intermediate format and *lparse*'s output language is not standardized) makes ASP solvers and related tools satellites of particular grounders, addicted to their capabilities and supported language fragments. We think that the establishment of an extensible intermediate language standard, not dictated by the capabilities of grounders, might motivate future works on knowledge representation for applications, inventing new language constructs when they are useful and then integrating them into the standard. At the moment, incorporating new language features would mean hacking one of the few available grounding tools, making the broad acceptance and usage of the feature rather unlikely. However, the establishment of an intermediate language standard must be a community effort, requiring a representative standardization committee and developers motivated to implement the standard in their tools. In view of these requirements, our proposal of *ASPils* can serve as a starting point for future discussions within the community.

Let us note that the establishment of *ASPils* or a comparable intermediate language standard can only be a small step in making ASP tools more general, more interoperable, and thus more user-friendly. In fact, tools are needed to perform various useful tasks on problem descriptions in the new language, similar to what the Helsinki collection (Janhunen) of tools offers for *lparse*'s output format. Let us give some examples. For the use in ASP system competitions, solver inputs must contain neither meta-information nor comments, thus, a (trivial to develop) tool to delete such information would be needed. For benchmarking, a tool

like *shuffle* is desirable, in particular, considering that shuffling *ASPils* sentences requires more care than needed with *lparse*'s output format as the order (Gebser *et al.* 2008a) among object definitions and references has to be maintained. If a grounder generates *ASPils* output on-the-fly, it is hard to predict the most restrictive normal form sufficient for the input program, hence, a postprocessor calculating this simplest normal form might be useful. As the last example given here, a tool like *lplist* should be made available for reconstructing a symbolic representation from the intermediate format. Finally, we note that *ASPils* or any other intermediate language cannot establish compatibility among the input languages of grounders or integrated ASP systems (such as *dlv*). Furthermore, modular (Oikarinen & Janhunen 2006), incremental (Gebser *et al.* 2008b), or even systems dealing with non-ground input programs are currently out of the scope of our proposal. However, we think that the relatively simple concept of an intermediate language provides a good basis for standardization efforts, and more sophisticated matters could be addressed in the future.

Acknowledgments. We are grateful to Marcello Balduccini, Martin Brain, Maarten Mariën, and Axel Polleres for fruitful discussions and valuable ideas that contributed to this work. Also, we would like to thank the anonymous reviewers whose comments helped us to improve the presentation. The second author was supported by the project #122399 "*Methods for Constructing and Solving Large Constraint Models*" funded by the Academy of Finland.

References

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*.

Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. *JACM* 7:201–215.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *CACM* 5:394–397.

Dell'Armi, T.; Faber, W.; Ielpa, G.; Leone, N.; and Pfeifer, G. 2003. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In *Proc. of IJCAI'03*, 847–852.

DIMACS. 1993. Satisfiability suggested format. [9]

Erdem, E., and Lifschitz, V. 2003. Tight logic programs. *TPLP* 3(4-5):499–518.

Fages, F. 1994. Consistency of Clark's completion and the existence of stable models. *JMLCS* 1:51–60.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007a. clasp: A conflict-driven answer set solver. In *Proc. of LPNMR'07*, 260–265.

Gebser, M.; Liu, L.; Namasivayam, G.; Neumann, A.; Schaub, T.; and Truszczyński, M. 2007b. The first answer set programming system competition. In *Proc. of LP-NMR'07*, 3–17.

Gebser, M.; Janhunen, T.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2008a. A versatile intermediate language for answer set programming: Syntax proposal. [10]

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2008b. Engineering an incremental ASP solver. In *Proc. of ICLP'08*.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. GrinGo: A new grounder for answer set programming. In *Proc. of LPNMR'07*, 266–271.

Gelfond, M., and Leone, N. 2002. Logic programming and knowledge representation — the A-Prolog perspective. *AIJ* 138(1-2):3–38.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *NGC* 9:365–385.

Giunchiglia, E.; Lierler, Y.; and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *JAR* 36(4):345–377.

Janhunen, T. The ASPTOOLS collection. [11]

Janhunen, T. 2007. Intermediate languages of ASP systems and tools. In *Proc. of SEA'07*, 12–25.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proc. of IJCAI'05*, 503–508.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM TOCL* 7(3):499–562.

Lierler, Y. 2005. cmodels - SAT-based disjunctive answer set solver. In *Proc. of LPNMR'05*, 447–451.

Lifschitz, V.; Tang, L.; and Turner, H. 1999. Nested expressions in logic programs. *AMAI* 25(3-4):369–389.

Liu, L., and Truszczyński, M. 2005. Pbmodels - software to compute stable models by pseudoboolean solvers. In *Proc. of LPNMR'05*, 410–415.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, 375–398.

Marques-Silva, J., and Sakallah, K. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE TC* 48(5):506–521.

Mitchell, D. 2005. A SAT solver primer. *EATCS* 85:112–133.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC'01*, 530–535.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *AMAI* 25(3-4):241–273.

Oikarinen, E., and Janhunen, T. 2006. Modular equivalence for normal logic programs. In *Proc. of ECAI'06*, 412–416.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *AIJ* 138(1-2):181–234.

Syrjänen, T. Lparse 1.0 user's manual. [12]

[9] ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/

[10] http://www.cs.uni-potsdam.de/wv/pdfformat/gejaosscth08a.pdf

[11] http://www.tcs.hut.fi/Software/asptools/

[12] http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz